MVC Design Pattern with PHP and SQlite

**Table of Contents**

[Introduction 2](#_Toc509234281)

[Part A — MVC Framework with PHP 3](#_Toc509234282)

[Building an MVC Framework using PHP 3](#_Toc509234283)

[Controller 6](#_Toc509234284)

[Model 8](#_Toc509234285)

[View 9](#_Toc509234286)

[Test the basic MVC framework 11](#_Toc509234287)

[Add images to the view 11](#_Toc509234288)

[Test the updated MVC framework 15](#_Toc509234289)

[Part B — Adding SQLIte with PHP to the MVC Framework 17](#_Toc509234290)

[SQLite 18](#_Toc509234291)

[Building an MVC Framework using PHP and SQLite 18](#_Toc509234292)

[Starting to use SQLite 19](#_Toc509234293)

[Add more methods to your controller 21](#_Toc509234294)

[Test your controller code 24](#_Toc509234295)

[Connect up the SQLite database 25](#_Toc509234296)

[Develop our views 33](#_Toc509234297)

[Test your new MVC code 35](#_Toc509234298)

[Some improvements 36](#_Toc509234299)

[Laboratory 8 38](#_Toc509234300)

# Introduction

The purpose of this Week 7 tutorial is to introduce you to the concept of a model, view, controller (MVC) design pattern or framework, which you will implement as part of your 3D App design. You will convert your Lab 6 3D App SPA into a MVC design pattern using this framework.

Unfortunately, it takes a lot more time to format code in these tutorials, and this tempts you to cut and paste for the word document, which can lead to errors. So, as usual, I will rely on screenshots of most of the code, and you should consult the **Live Feedback Site** for a working version of Lab 7. However, as the back end will be written in PHP, which is on the server-side you won’t be able to see the actual PHP code!

So, as we are at week 7, and many of you will be catching up, you should not forget that all lab tutorial codes are available for download on GitHub, see Part 6: Set up Git and GitHub with Visual Studio Code in the Lab 1 tutorial. You will then be able to cut and paste, where appropriate. However, to download, you will need a GitHub account, unless you are happy to type all the code in again from the screen shots! But, do not just download it and expect it to run, you will have to integrate it into your Lab 6 results! You can download the tutorial code from https://github.com/martinwh/3D\_Apps\_2019 in the Lab 7 folder.

Having two screens while completing this tutorial is very useful, because you can have the tutorial text open, the Lab 7 MVC code from GitHub in a separate code editor, so that you can see it as you read this tutorial, and then develop in another code editor — or use a code editor that has more than one window.

# Part A — MVC Framework with PHP

Part A of this tutorial explores the development of a simple MVC framework written in PHP to implement our 3D App SPA.

The ‘3D Apps — Technology Stack’ section on MVC, AJAX, JSON, and Data Stores (see Canvas) discusses MVC in more detail. During this laboratory, we will be developing our own very simple MVC framework.

So, through this tutorial and some background research you should become familiar with the Model, View, Controller (MVC) design pattern. You are about to develop this simple MVC framework using the following technology stack: HTML5, CSS3, JavaScript, JQuery at the front-end connecting via jQuery and AJAX to a PHP and SQLIte back-end — along with libraries such as Bootstrap 4.

There are, of course, [other variants of MVC](https://lostechies.com/derekgreer/2007/08/25/interactive-application-architecture/), but we will stick with MVC for this Lab 7 Tutorial. Similary, Instead of building our own PHP based MVC framework, there are many web frameworks (both PHP and Java based) we could choose from, but it is more instructive to develop our own simple version so that you can better understand the concepts. You can explore and experiment with other MVC frameworks, such as [CodeIgnitor](https://codeigniter.com/), [Laravel](https://laravel.com/) and [SLIM](https://www.slimframework.com/) (a micro framework for PHP), but it is more instructive to develop our own simple version so that you can better understand the concepts.

In this tutorial, we will use the MVC design pattern whereby a view makes a request to a **controller**, which then invokes a **model** to retrieve data from a data store (or model, which could be a JSON file, SQLite database, or a third-party API endpoint) and passes a response back to the controller, which then updates the **view** to publish the response (e.g. your 3D App data) on a web browser.

This tutorial will demonstrate how easy it is to replace a view from a website without any modifications to the model and virtually no modifications to the controller, which is possible with MVC design patterns as it effectively separates out the data (or model) from the business logic (the control) and the web visualisation (or view).

Note, we will not simply create bits of PHP code to access your 3D App data from the SQLite database, neither shall we use procedural PHP. Instead, we will use classes and objects in PHP (later versions of PHP are object oriented), and although these are just simply introduced here some background reading may be needed if you are not familiar with [object oriented PHP](https://www.killerphp.com/tutorials/php-objects-page-1/). Here is a very short, but nice example of [procedural PHP versus object oriented PHP](https://gist.github.com/sycue/7382545) on GitHub.

Further, we will also introduce the notion of *data-access abstraction* using PDO (PHP Data Objects). The [PHP Data Objects](http://php.net/manual/en/intro.pdo.php) (PDO) extension defines a lightweight, consistent interface for accessing databases in PHP.

## Building an MVC Framework using PHP

During this tutorial, you will learn how to build your own extremely lightweight PHP framework based around the MVC (model-view-control) architecture or design pattern using object oriented PHP.

We will use a step by step approach:

1. Start by creating a new folder for Lab 7, e.g. call it Lab 7, whatever. However, unlike before don’t copy over your Lab 6 yet, up to now we have been evolving the previous week’s lab into the next one. Instead, we will start to create the model and view components of the MVC framework. Your view(s), will of course be the based on your current index.html template for lab 6, however this will become a view in the view folder of the upcoming MVC folder structure. As before, you will need a live website environment to test your new MVC based 3D App, for example:

* You may already be using a localhost environment on your own PC or laptop, e.g. WAMP, MAMP, XAMPP or even the Mac built-in apache web server.
* You may instead be using your public\_html environment as your Local Folder (if you haven’t got this set up yet — I noticed in Lab 6 some of you still did not have this set up — simply do a search on Google for something like: ‘Sussex ITS Web Space’ and follow the instructions to set up your ITS web space.

1. Create a file named as ‘index.php’ using any text editor, e.g. Notepad++, Brackets, Sublime Text (I am currently using Visual Studio Code, for which I have already discussed in lab 1 that it is easy to integrate with [GitHub — a version control repository and hosting system](https://github.com/).

* Now type the following lines of code in ‘your index.php’ file, Figure 1.
  + Note, when I say ‘type’, you can of course type, but if you have downloaded the tutorial code, as indicated at the start, from GitHub, you can simply ‘cut and paste’.

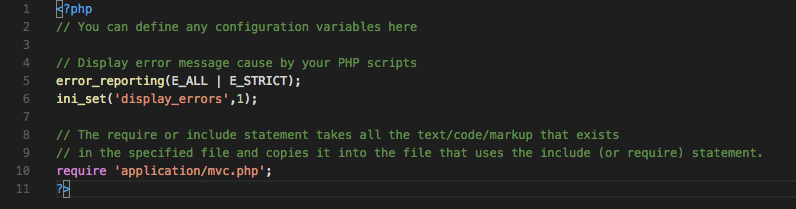


Figure 1: The index.php file

* Code explanation
  + Line 1 and 11: ‘<?php’ denotes the start of the PHP code, and ‘?>’ denotes the end of the PHP code.
  + Anything after ‘//’ is treated as comments, and these are only used for code readability. The PHP compiler (embedded in the browser) will simply ignore the whole line, starting from the ‘//’ characters.
    - We have added these some comment lines to show how to use comments in your PHP code.
    - Line 5 and 6: The PHP library function ‘error\_reporting()’ and ‘ini\_set()’ enables debugging of your code, i.e. any errors generated will display when your PHP script has any bugs. If you remove this, you will not be able to see any errors in your code and your browser will block the site in the case of any error in your code — you will simply be left scratching your head wondering where the bug is! So, it is very important that you keep these functions in your code to see any error reporting during PHP code development. As the PHP functions we will develop are relatively simple we will manage with this error reporting, but for anything more sophisticated you might want to introduce other [PHP debugging options](http://php.net/manual/en/debugger.php), which range from printing out variables, building a console\_log function, to including a debugging library.
    - Line 10: The require 'application/mvc.php'; tells the browser to load or include a file named ‘mvc.php’ from the ‘application’ folder into the current file.
  + Effectively, this index.php file is acting as a configuration file to define global variables and URL includes, since this file is the starting point of our site.

1. Create a folder called ‘application’ inside your ‘’ folder
2. Then create three subfolders inside the ‘application’ folder, called: controller, model, and view

* Your folder structure should now look like that shown in Figure 2.

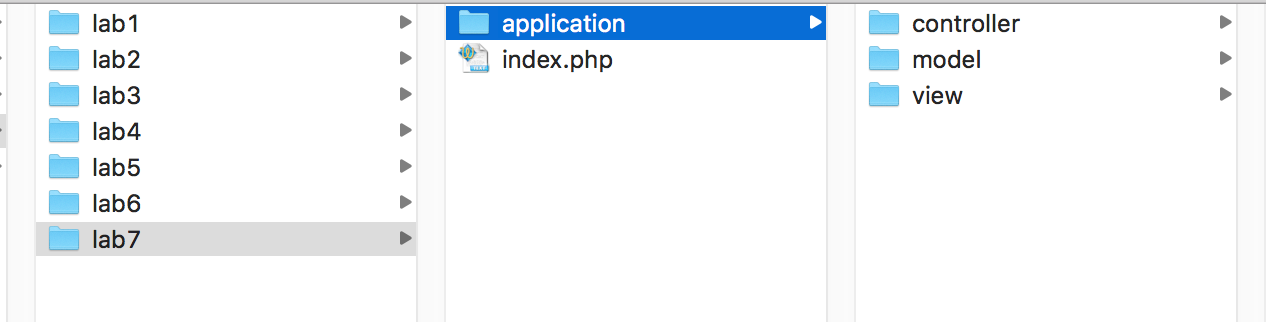


Figure 2: Set up for the MVC folder structure on a Mac, should be a similar structure on a PC

* As you can see, we now have three different folders to hold your application’s ‘model’, ‘view’ and ‘controller’; we will make sure that we store:
  + The data access code inside the ‘model’ folder — this will be a model class with methods to access the data store, e.g. the SQLite database
  + Front end presentation code in the ‘view’ folder — this your existing index.html (Lab 6) file, but we will rename it view.php, or something like that.
  + And, business logic or control code in the ‘controller’ folder — this will be a PHP class with methods that invoke your model class methods
    - Your getJsonData.js file will be re-written to make AJAX calls to these PHP methods in the controller using the JQuery .getJSON() function.

1. Before we create code inside those model, view, controller folders, Figure 2**.**, let’s create an entry point for our application. This PHP file will refer to all three components, model, view and controller and it will initialize (create an instantiation) the controller of our application.
2. Create a PHP file inside the ‘application’ folder and name it as ‘mvc.php’

* Now type the following code in it and save the file, Figure 3.

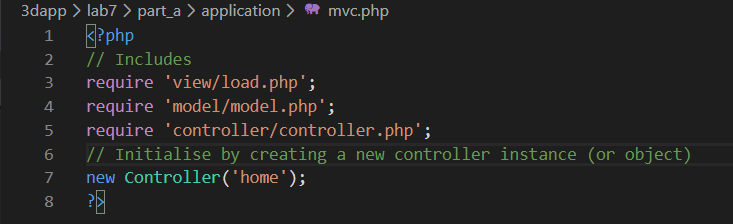


Figure 3: The PHP entry point file — mvc.php

* Code explanation
  + Lines 3, 4 and 5 are code to include the three different files from the three folders (model, view and controller folders), these statements will include three PHP files (load.php, model.php and controller.php). This means from this point we can use any function or variables from within these three files (load.php, model.php and controller.php) in our application.
    - Just as we included with the ‘require’ in the index.php the mvc.php file so at this level inside the mvc.php we include other php files.
  + Line 6: Initializes the controller, by creating its object / instance.
  + Note we didn’t call the file in the view folder view.php. Why do you think that is? It is because we might have many views and we will use the load.php file to select and load the views.

### Controller

Now let’s create our core components of the MVC framework.

1. Create a PHP file inside the ‘controller’ folder and name it as ‘controller.php’
   * Create the following code in the controller.php file, see Figure 4.

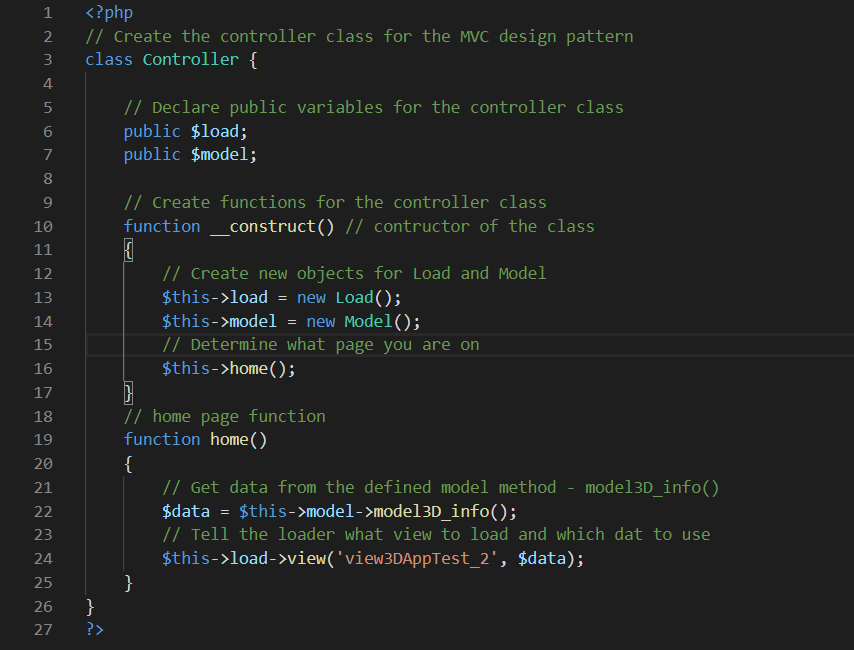


Figure 4: The PHP controller file

* Code explanation
  + Lines 2 to 23: defines a class called ‘Controller’. Everything between parentheses ‘{‘ and ‘}’ is the body of the class.
    - Class definition: PHP gives us a very simple way to define an object as a class, which may contain its own constants, variables (called properties) and functions (called methods). A class is effectively a wrapper around the object that defines and encapsulates the object’s properties and methods. The class represents the PHP interface that allows you to interact and modify the object. You can create multiple instances/objects of one class, each object has separate memory space and has a separate copy of all members defined in that class, hence is reusable. A class can have many or no methods, we usually use these methods to set, modify and get properties from the object.
  + Lines 6 and 7: Defines ‘load’ and ‘model’ variables, and since they are declared as ‘public’, they can be accessed from anywhere using the instance of the class.
  + Lines 10 to 17: We define a constructor of the class, again everything between parentheses ‘{‘ and ‘}’ is the body of constructor.
    - Constructor definition: A constructor is the first function that is executed when a class is initialized. This means anything in constructor will execute immediately as soon as we create an instance of the class.
    - In this example, the constructor function creates new objects for load() and model(), see below, and then calls the home() method (or function), which gets the car data from the model3D\_info() model and presents it in the view3DAppTest view.
    - The two underscores before the constructor method indicate that PHP is treating this construct method as [magical](http://php.net/manual/en/language.oop5.magic.php), you should not use a double underscore for any of your own methods/functions.
    - With OO PHP, all PHP objects (classes) have a special built-in method called a constructor that allows you to initialise your properties, i.e. give your properties values, when you instantiate, i.e. create, an object. Note that: functions = methods, and variables = properties
    - Note, if you create a constructor, and you don’t have to, but it is very useful to initialise properties, the PHP will automatically call the \_\_construct() method/function when you create an object from your class.
    - The constructor can be by passed by specifically calling the Home function in a URL?
    - ‘$this’ refers to the current object of the class, so that

$this->load and $this->model

means we are accessing the ‘load’ and ‘model’ variables from the current class objects instantiated with the new Load() and new Model().

* + - ‘new Load()’ defines a definition of new object of the class named ‘Load’.
    - Similarly ‘new Model()’ defines a definition of new object of the class named ‘Model’. Note we have not written the Load and Model classes yet; we will define these two classes inside ‘model’ and ‘view’ folders below.
    - ‘$this->home()’ refers to the home function from the current class object.
  + Lines 19 to 25: contain a definition of the ‘home’ function.
  + Line 22: ‘model3D\_info()’ is a call to a function from the model class for the current object. The function model3D\_info() returns value(s) that is/are stored in a test ‘data’ variable or array.
  + Line 24: similarly ‘view(‘view3DAppTest’, $data)’ is a call to the function named ‘view’ from load object.
    - ‘view3DAppTest’ and $data are the parameters passed to the view function.

### Model

We now need to create some model code to simulate the database contents.

1. So, create another file called ‘model.php’ inside ‘model’ folder, and type following code in it, see Figure 5.

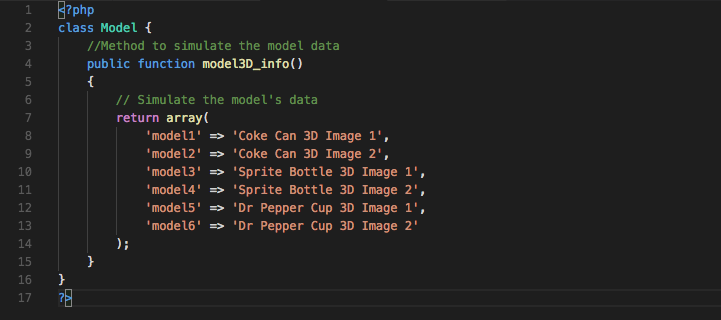


Figure 5: The PHP Model file

* Code explanation
  + Lines 2 to 16: defines a class called ‘Model’. Everything between parentheses ‘{‘ and ‘}’ is the body of the class. You can immediately see that we simply simulating a data store by populating an array with data to return to the calling function.
    - Recall: a class is a wrapper around the concept of an object, which contains variables (properties) and functions (methods). A user can create multiple instances/objects of one class, and each object has separate memory space and also has a separate copy of all members defined in that class.
    - We have not written a constructor method for Model, so the function model3D\_info() will default to the construct method, that is when you instantiate this object Model, the function model3D\_info() will execute.
  + Lines 4 to 15: contains definition of the function named ‘model3D\_info’
  + Lines 7 to 14: defines an array, which is not stored in any variable at this stage but its reference is returned to the function calling statement.
    - This array has five values (Coke Can 3D Image 1, …) and five pointers (model1, …) to those values.
    - Later, this array will also hold data for some images, which we will with a second view to illustrate the loading of different views using data from the model.

Note: At this stage, we are just simulating a database in the model class. We can use this class to communicate with an actual database to retrieve any data; once we have the data from database, it can then be passed in similar way to ‘view’ class to publish on a browser.

### View

Now let’s create our view. Since in this tutorial we are going to create two different views for the same class, we need two different HTML files and one intermediary load.php file to select between those two views according to the instructions provided by the controller.

1. So, create a file named ‘load.php’ inside the ‘view’ folder. You can begin to see now why we called this file load.php and not view.php. The role of load.php will be to load a view, where a view will be an html file.
   * Now type the following code in the load.php file, see Figure 6

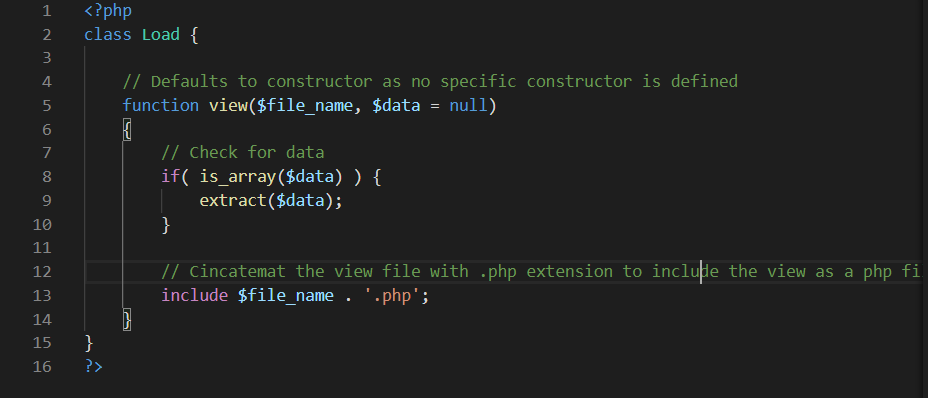


Figure 6: The PHP load file

* Code explanation
  + Lines 2 to 15: defines a class called ‘Load’.
    - As before, the function view defaults to the construct method, because we haven’t specifically created the \_\_contruct() method.
    - Everything between parentheses ‘{‘ and ‘}’ is the body of the class.
  + Lines 3 to 14: defines a function (or method) called ‘view’.
    - Everything between parentheses ‘{‘ and ‘}’ is the body of the function.
  + Line 8: the ‘if’ statement checks the ‘$data’ variable to see if it is a pointer to an array? If it is a pointer to an array, the program control goes to line 9, if it is not pointer to an array the program control will skip line 9.
  + Line 9: a predefined php function named ‘extract’ is used to import all variables from an array into the current file, so that our view can use them as variables rather than array elements.
  + Line 13: includes value of the variable $file\_name provided as an argument to view function.
    - **Remember**: in the controller class line number 24, we passed the ‘view3DAppTest’ string to the ‘view’ function, so in above case the value of the variable ‘$file\_name’ is ‘view3DAppTest’.
    - Succeeding the variable $file\_name there is a ‘.’ character, which is used for concatenation of two strings. In above case the value of $file\_name (which should be ‘view3DAppTest’) is concatenated with ‘.php’ string, and the resulting string becomes ‘view3DAppTest.php’.

1. Create another file, save as ‘view3DAppTest.php’ also inside the same ‘view’ folder, and type the following HTML code in it, see Figure 7.

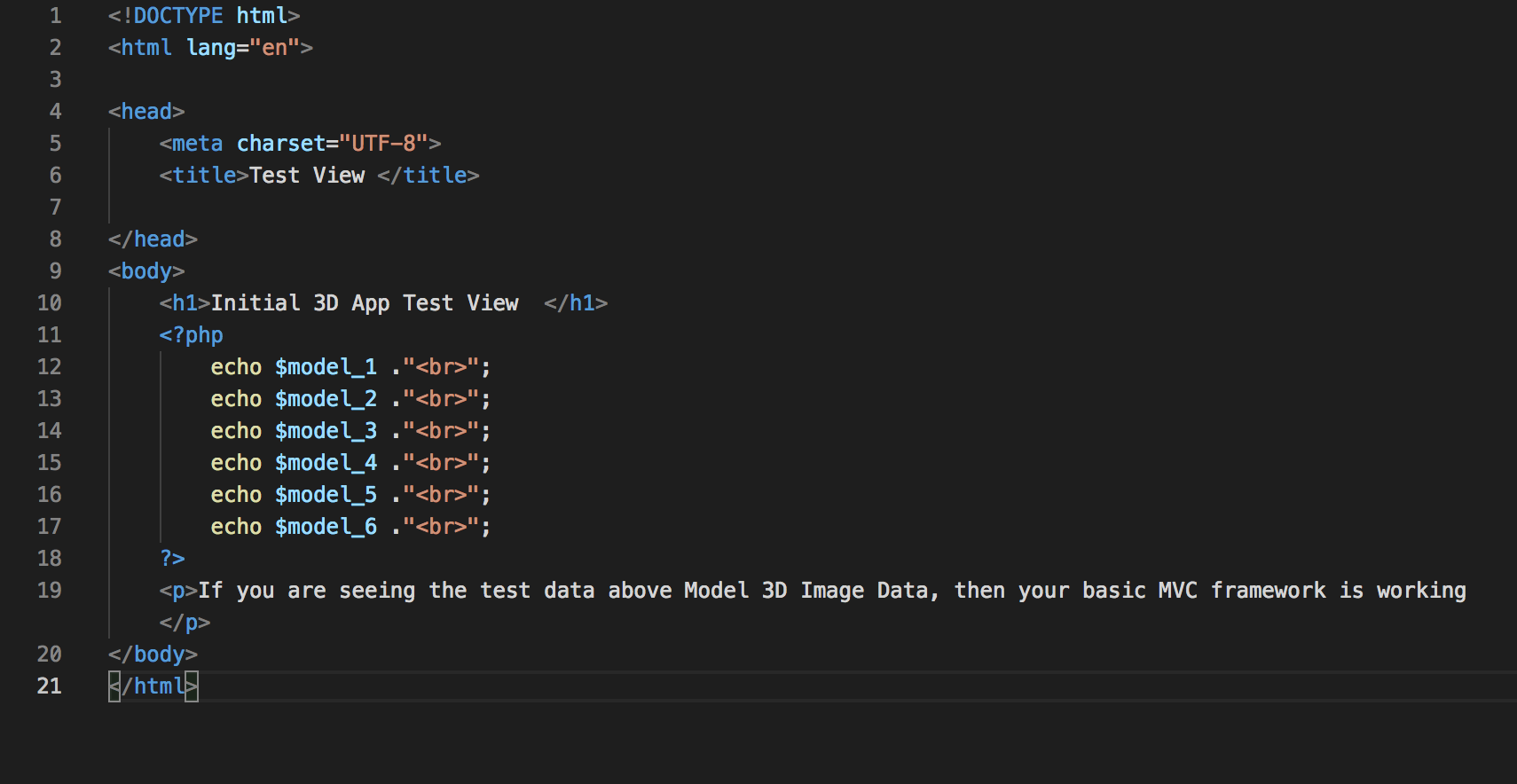


Figure 7: The PHP view3DAppTest file

* Code explanation
  + The code above is mostly HTML code except between lines 9 to 15. You should already be familiar with some simple HTML code, so we won’t explain this in more detail.
  + Lines 12 to 17: the PHP statement ‘echo’ is used for displaying a text on the browser. In the above case, the values of variables $model1 to $model6 are displayed on the browser and after each output a line break is inserted by concatenating ‘<br>’ in the echo statement.

## Test the basic MVC framework

1. Now let’s execute our code, by typing our website url into any internet browser, see Figure 8.

* Note: You cannot simply double click on a php to run in a browser, you will need a php server to see its output. You should have been provided a PHP hosting space by ITS, which should be your ‘public\_html’ folder.
* Alternatively, use a localhost on your own PC or laptop.

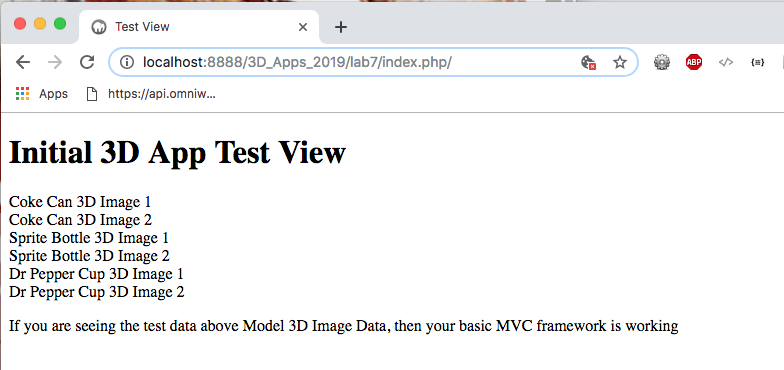


Figure 8: The HTML output generated by the view

Note: the url above is ‘http://localhost:8888/’, since I am running on my Mac’s localhost, but it will be different in your case. If you are using your public\_html folder on your ITS hosting space, your URL should be something like this:

* users.sussex.ac.uk/~your-user-name/3dapp/lab7 /index.php
  + In the above output, the data comes from the **model** and is displayed in the **view**, and the **controller** has coordinated between these two components.

## Add images to the view

In this section, we will use some of your 3D images as data. We will create a new view called view3DAppTest\_2.php, and we will update our model test data to reference the images.

Carry on following the steps below:

1. Make a copy of your ‘assets’ folder from Lab 6 and past it into your Lab 7 folder. Now, your folder structure should look something like that shown in Figure 9.

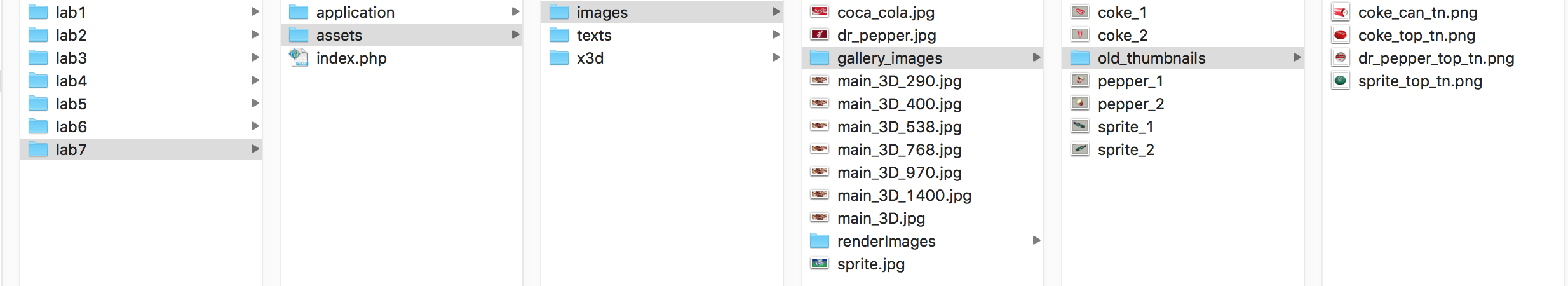


Figure 9: Your folder structure

Note: Your assets folder may be organised differently to mine, what we need are 6 images to join with the text data test data we used above. I simply created 6 screen shots (could have been 3ds max 2017 rendered images) of my coke, sprite and pepper models in my Lab 6 3D App — so these will do.

1. Next, if you didn’t update yet your model.php file, see Figure 10.

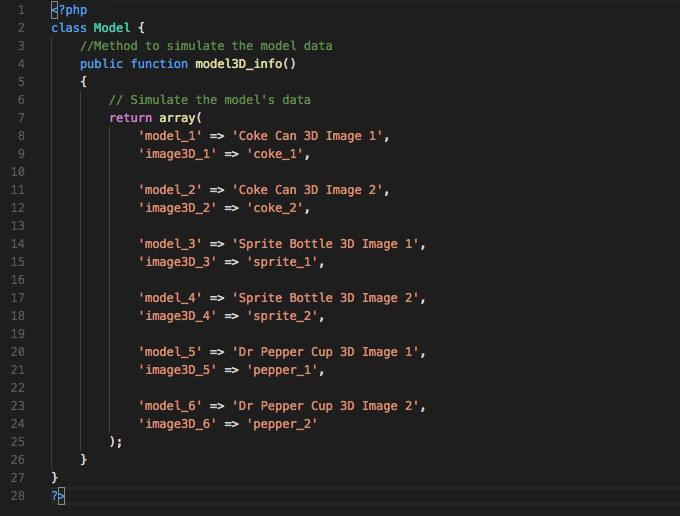


Figure 10: Update your test model.php file

* Code explanation
  + You can see that all I have done is added some more data variables whose values are the names of the stored images in the assets folder. You can see that I have left off the image extension, e.g. .png, .jpg. I could just as easily have put the complete image name here because it is a string. But, if you recall back in a previous lab where we created the gallery generator code, we would normally read all file names in the folder and process the extensions anyway.

1. Now create a new view test file named ‘view3DAppTest\_2.php’ inside the application > view folder, with appropriate code to join your text test data to your 3D image data, see Figure 10.

* Code explanation
  + In this view, you can see that I have just created a set of 6 div tag block to hold each of the image data groups, i.e. the image title and image file
  + Further, we have created some CSS rules to style the data.
    - The .box class provides a block element for the data that floats left so that each image and its associated test data floats around each other to the left as the window is resized.
    - The .imgBox class styles the image inside the box. I just robbed up some CSs here from the custom .css file in Lab 6.

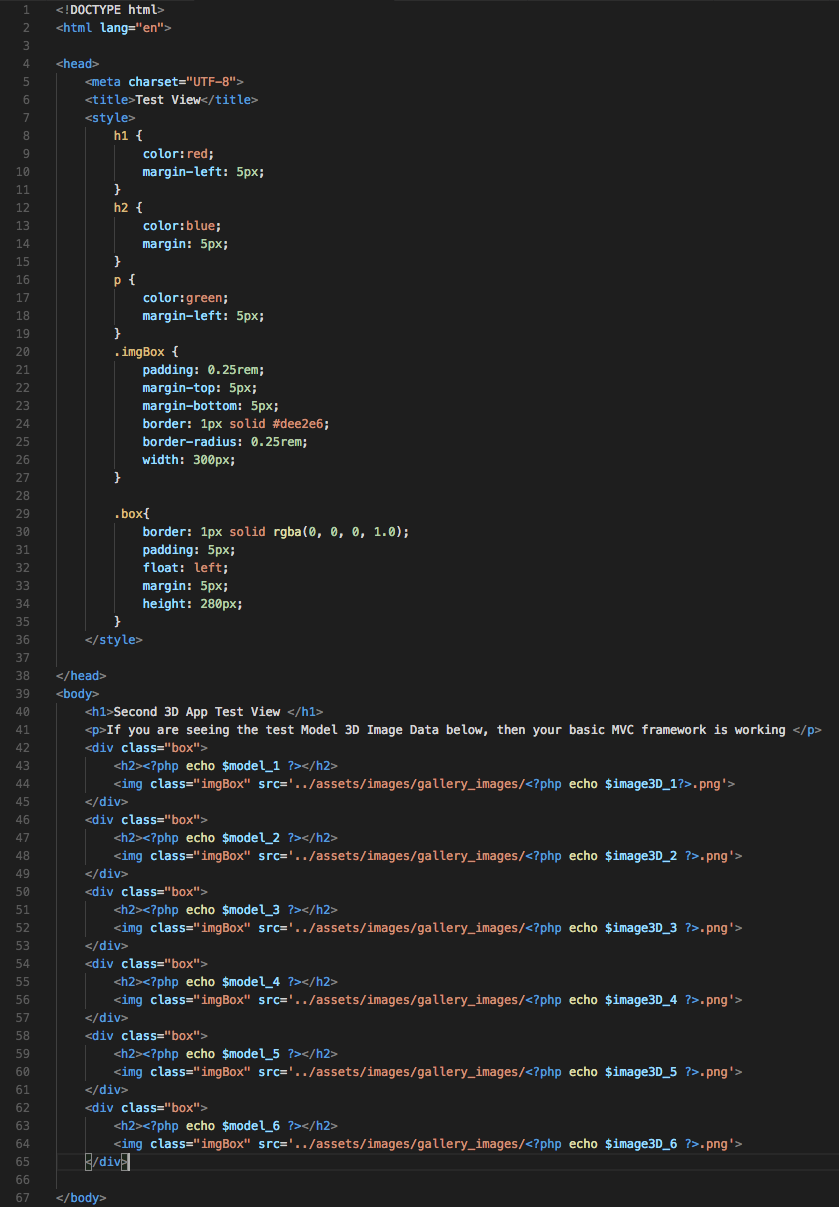


Figure 10: The view3DAppTest2.php file

1. All we need to do now is change name of the view in controller.php, see Figure 11.

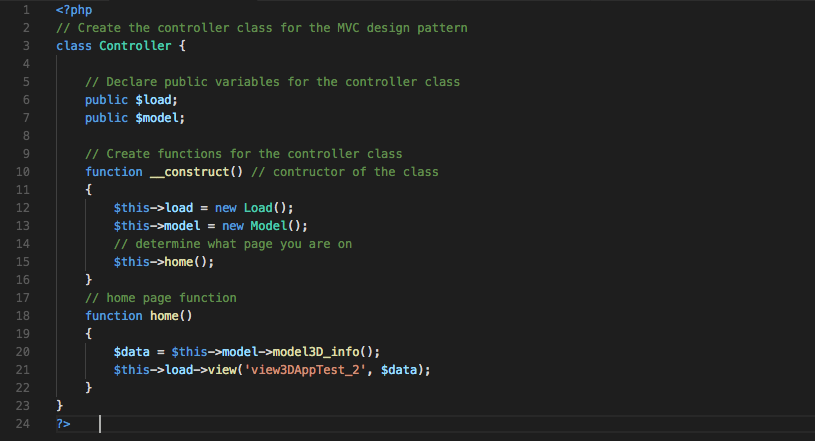


Figure 11: The new Controller PHP file

* Code explanation
  + This is the same controller from last example, where we have edited the controller.php file to change the first parameter in the view function from ‘view3DApptest’ to ‘view3DApptest\_2’ on line number.

## Test the updated MVC framework

You can now test the updated MVC framework, see Figure 12. Note: Again, the url is: ‘http://localhost:8888/3D\_Apps\_2019/lab7/index.php/’, since I am running on a localhost on my Mac, but it will be different in your case. If you are using ‘public\_html’ folder on your ITS hosting space, your URL should be something like this:

* users.sussex.ac.uk/~your-user-name/3dapp/lab7/index.php

You should take the time to read around PHP, and particularly the MVC design pattern or architecture approach. There are formal MVC frameworks already built that you can exploit for both Java (e.g. Struts) and PHP (e.g. Codeignitor). Do a Google search and explore more. You will be expected to adopt this approach in your Assignment (hand in week 12).

Although we are using images so far, you should be able to see that you could just as easily organise the path to your stored 3D models in the backend so that they are rendered in the front end like the images. You would just need to organise the appropriate HTML for instantiation the X3D code at the front end, see Lab 6.

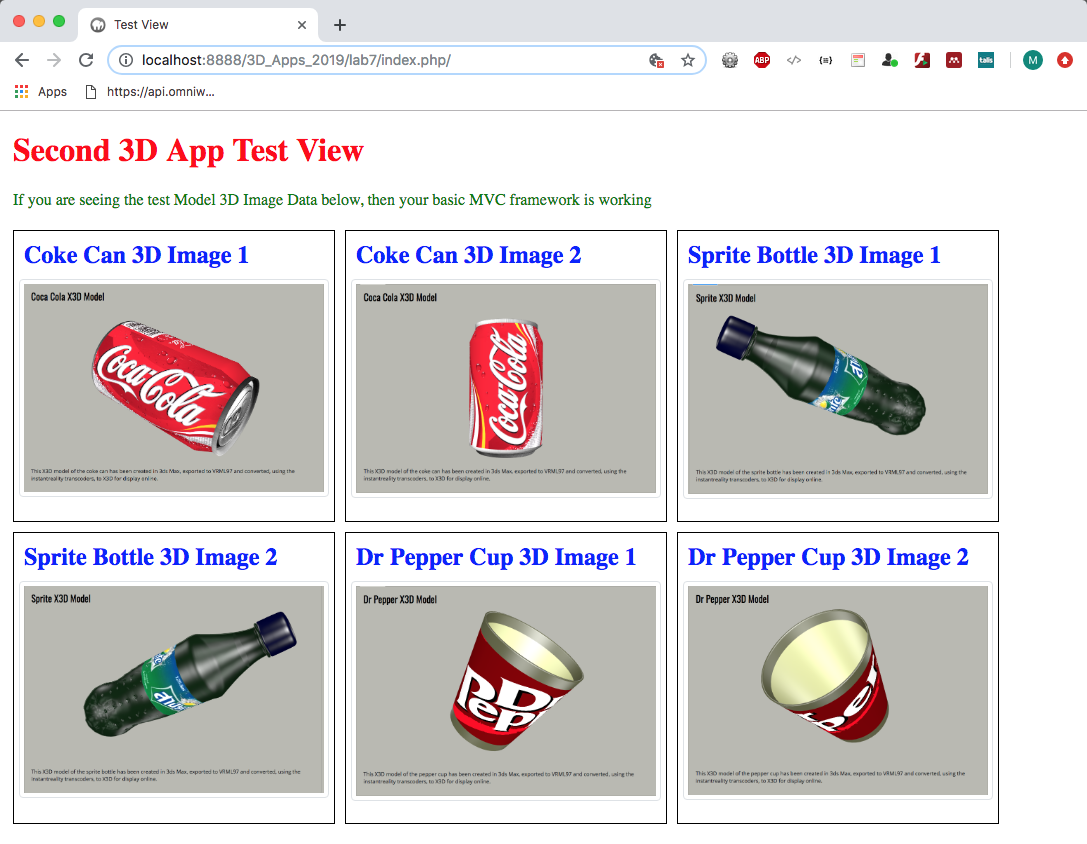


Figure 12: The second 3D App test view

# Part B — Adding SQLIte with PHP to the MVC Framework

This tutorial requires a very basic knowledge of PHP programing and MVC frameworks. If you have completed Part A of this tutorial then go ahead and complete Part B, but if you haven’t finished Part A, then we strongly recommend that you complete this now because Part B will be building on the source code we developed in Part A.

Part B of this tutorial describes how to use SQLite with PHP in your MVC framework. SQLite is a very convenient executable database, more precisely:

“SQLite is an in-process library that implements a [self-contained](https://www.sqlite.org/selfcontained.html), [serverless](https://www.sqlite.org/serverless.html), [zero-configuration](https://www.sqlite.org/zeroconf.html), [transactional](https://www.sqlite.org/transactional.html) SQL database engine. The code for SQLite is in the [public domain](https://www.sqlite.org/copyright.html) and is thus free for use for any purpose, commercial or private. SQLite is the [most widely deployed](https://www.sqlite.org/mostdeployed.html) database in the world with more applications than we can count, including several [high-profile projects”.](https://www.sqlite.org/famous.html)

As mentioned, to become more familiar with MVC concepts we are creating our own MVC framework, and we envisage the need to make it dynamic and capable of storing local data. There are many ways to store data, for example we could use the JSON file approach that we used in Lab 6. There are other solutions, for example we could use a flat file system and store our data in XML files, build an XML parser, and so on the retrieve the data, or we could use AngulaJS to bind data from a file to elements in the web page, we could even use Google’s Firebase.

But, a more flexible way, and one where we can make queries on that data using, for example SQL (Structured Query Language), is to use a database. To meet this need we will connect with the aforementioned SQLite database using PHP. Our PHP code to do this will largely cover several functionalities:

* Connection to the database
* Creating tables in the database
* Inserting data into the database tables
* Retrieving data from the database tables
* Deleting data from the database tables
* Closing the connection to the database

We will also introduce [PHP Data Objects](http://php.net/manual/en/intro.pdo.php) (PDO) as a data-access abstraction, “which means that, regardless of which database you're using, you use the same functions to issue queries and fetch data”[[1]](#footnote-1).

## SQLite

Why use SQLite as opposed to any other database? Putting aside the description of SQLite above as a good enough reason, we could consider using the University web server’s MySQL database, but that is impractical for teaching purposes. I think the lab machines now have a local host installed, so we could use an installation of a localhost with its MySQL (or SQLite) installation. But, it is easier to use your ITS Web Space, which is already set up for PHP and SQLite.

SQLite is a relational database management system developed in the C language. This is an extremely small size database management system — roughly about 350 Kbyte in size. Every time we create a new database using SQLite, it generates a small portable executable file that can be copied over to a different location.

This means we can locate our SQLite database, effectively a C executable, in your public\_html space (or on your local host).

### Building an MVC Framework using PHP and SQLite

Recall from Lab 7 Part A, where we created your MVC framework, let’s continue using the same framework to progress further in terms of database operations.

However, if you want to preserve your part\_a code intact, then you can copy all that code to a new folder below Lab 7, e.g. create a new folder, and call it part\_b, then move all your part\_a code structure to your now part\_b folder. You can then create a copy of your new part\_b folder and call it part\_a, which you will keep as a working archive folder and continue to update the part\_b codebase. That way if you mess up some code in part\_b you can always track back to a certain extent. Test your part\_a codebase still works!

Continue with Part B of this tutorial by completing the following steps:

1. First, let’s confirm that we have SQLite installed with our PHP installation. To do this, create a PHP file inside your public\_html folder in Lab 7 and name it ‘phpinfo.php’ — I am assuming you are developing on the ITS Web Server. If you are using a localhost, put the phpinfo.php file wherever you have designated the root your site to be, e.g. lab 7.
2. Now copy following three lines of in it, Figure 13:

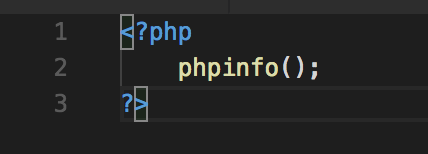


Figure 13: The phpinfo.php file for running the phpinfo() function

* Let’s see its output by typing its URL into a web browser, e.g. <http://localhost/3dapp/lab7/phpinfo.php> you may have a different path.
* If you are using your University web space, the host path will be different (users.sussex.ac.uk/) and need to incorporate your username (~username/) and path, e.g. (*path*/3dapp/lab7/part\_b/phpinfo.php) or wherever you have your files located, of course, see Figure 14.

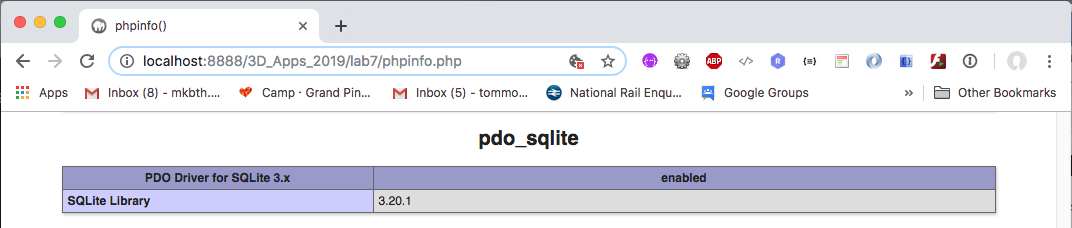


Figure 14: Check the PDO driver for SQLite3 is enabled

1. On the page above, search for ‘sqlite3’ section and if you don’t find ‘SQLite3 support’ set to ‘enabled’ that means you need to enable your SQLite before working with SQLite. To do this Google ‘how to setup SQLite in PHP’. You can only setup SQLite by yourself on your own machines; you cannot set this up for ‘public\_html’ folder provided by University. It should already be setup by the ITS for you. For Mac users, I tend to install MAMP, which already has SQLite3 enabled. On your own Windows PC or laptop, you might use WampServer, XAMPP, or something like that.

### Starting to use SQLite

1. Let’s start by creating a new folder inside your part\_b (i.e. at the same level as the application folder) folder and name it ‘db’. Your folder should look something like that show in Figure 15. This folder will contain the SQLite database. Do not mistake this for the model folder.

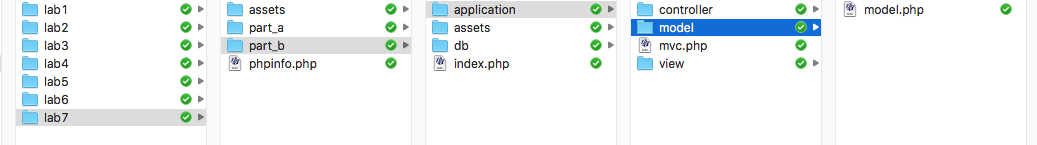


Figure 15: Adding the db sub-folder to your MVC structure

1. On the ITS web server, i.e. in your public\_html folder, you will need to change the read and write permissions on your db folder

* You have created a ‘db’ sub-folder in your folder, but this folder is not script write-enabled on the ITS web server at the moment, i.e. you can read any data from this folder, but you can’t write anything through a PHP script.
* To write enable this folder you will need a terminal so that you can run a couple of commands: touch db/ and chmod o+rw db/.
* If you are working remotely you could download or access an open source terminal emulator, such as [PuTTY](https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html). If you are in the labs, PuTTY is already installed, I think.
* But, you should just be able to navigate to your db folder anyway, and then bring up a terminal window to change the db folder permissions.
* However, assuming you are remotely changing permissions on your db folder, you need to download putty, login to the unix.sussex.ac.uk server and assign read write permissions to it. Follow the steps below to assign read and write permission to your ‘db’ folder on your public\_html folder.
* Open up Putty and type as shown in the image below, see Figure 16. That is, the Host Name: unix.uscs.susx.ac.uk, Port: 22 and then Click open

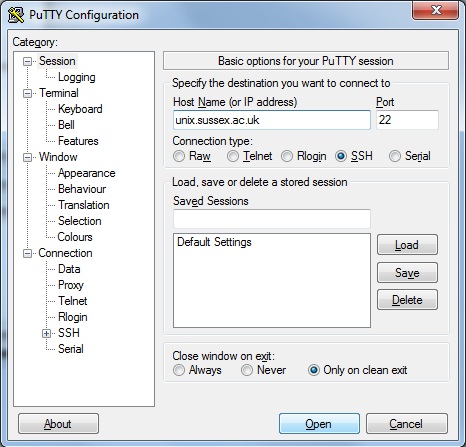


Figure 16: Putty interface

* On the shell screen, see Figure 17, you will be asked to enter your login user name, enter your login user name.
* Do the following commands:
  + Navigate to your part b folder, e.g. cd public\_html/3dapp/lab7/part\_b/. Or wherever you have your ‘db’ folder located. Note, in Figure 17 /tiny\_mvc/ was an old folder from previous years.
  + touch db/
  + chmod o+rw db/
  + exit

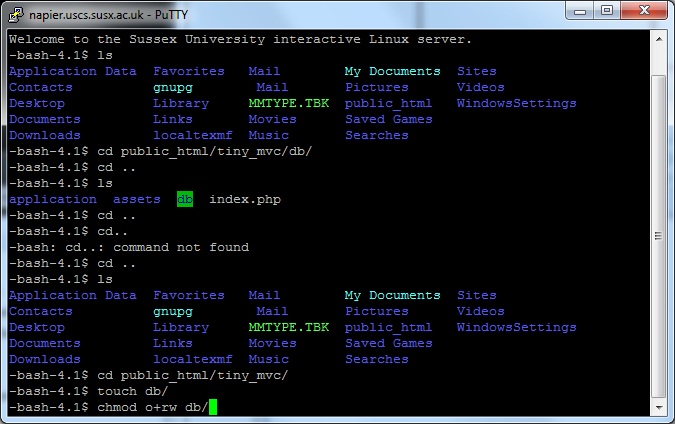


Figure 17: Changing db permissions to allow read and write

This applies to your work in lab 7 and other labs where you are creating an SQLite database, in which you need to read and write data. If you are working on a localhost, you should be ok. However, you’ll need to do this when you port to public\_html.

Note, if you have been working with a localhost on your own PC or Laptop, your db folder will probably already have the right permissions before you upload to the ITS Web Server.

## Add more methods to your controller

1. Since, we are now going to have many controllers in our example, we need to identify the names of these controllers from the browser’s URL.

* We already created two views: 1) view3DApptest.php and 2) view3DAppTest\_2.php. In this case, we simply edited the controller method ‘home’, i.e. the function home() in the controller.php file. What we could have done is create two new methods, one for each of the two different views.
* We would then access these two methods by appending their controller function or method name to the end of the URL, so it is useful to give the controller function or method a useful name such as, get3DAppData, or in this case view3DAppTest\_2.
* Thus, such a method would be accessed via the following URL, for example: http://localhost:8888/lab7/part\_b/index.php/get3dappdata. BTW, this URL won’t work because we don’t have a method called get3dappdata yet,
* However, we would also need to modify our mvc.php file to capture the method appended to the end of the URL, see Figure 18. Implement this code.

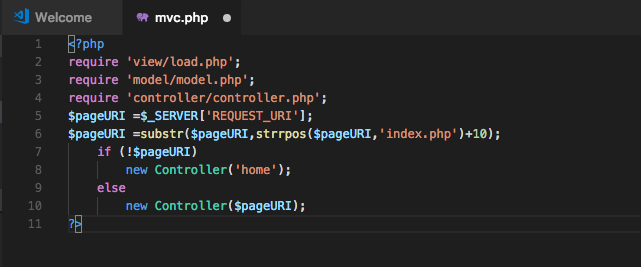


Figure 18: The new mvc.php code

* Code explanation:
  + Line 5: ‘pageURI’ is assigned the value of the URI (Uniform Resource Identifier)
    - For example: $SERVER[‘REQUEST\_URI’] will return /lab7/part\_b/index.php/view3DAppTest\_2  
      from the URL http://localhost:8888/lab7/part\_b/index.php/ view3DAppTest\_2.
  + Line 6: ‘pageURI’ is now assigned the value (i.e. the name) of the controller method (function — in this example: view3DAppTest\_2) from the URI using the substring function.
    - To be clear: The substring function (substr()) above will return ‘view3DAppTest\_2’ from the URI: ‘/index.php/view3DAppTest\_2’. It does this by using the strrpos() function to find the position of the last occurrence of index.php in the URI stored in the variable $pageURI and then adding 10 (the number of characters in index.php/ to find the starting character of the controller function (method) being called, which in this example was view3DAppTest\_2.
  + Lines 7 to 10: Here, we make a decision, if the ‘pageURI’ variable gets nothing from line 6 (for example, when you only enter ‘…/index.php/’ so that you have not entered a name of a controller method to call), then the default controller method (\_\_construct remember) is called with parameter ‘home’. Clearly, if the variable ‘pageURI’ is not empty then a controller is called by the name set as a value of the ‘pageURI’ variable.

1. Now let’s modify our controller class to deal with new parameters. Modify your controller.php file inside ‘/application/controller’ folder as shown in Figure 19.

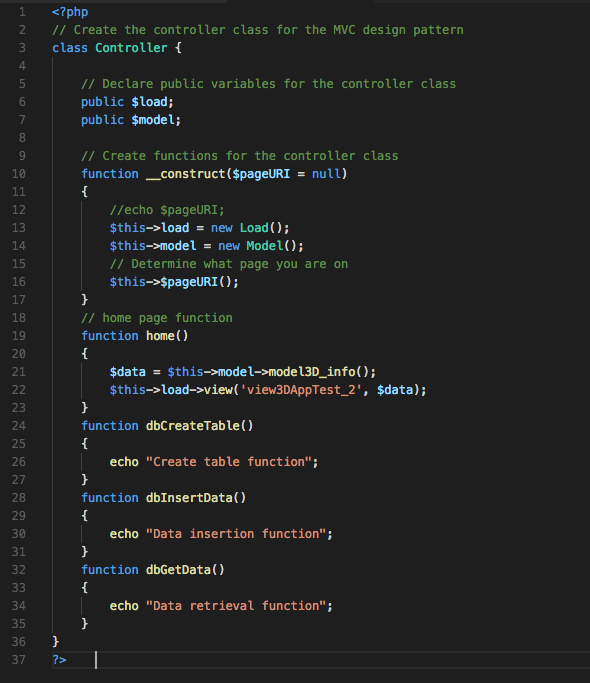


Figure 19: The new controller php code

* Code explanation:
  + Line 10: We have now added a ‘pageURI’ input parameter to our \_\_constructor.
    - This is will receive the name of the controller from ‘mvc.php’ file.
  + Line 16: $this->home is now changed to $this->$pageURI
    - This means, we are now calling a function (method) name identified by ‘pageURI’ variable. Before this was pointing to ‘home’, because previously, in part a, we only had the ‘home’ function, but now we have many functions (or methods) inside the controller class —we actually had two sets of home page contents, if you recall, but we didn’t have a way of selecting them, which we are doing now.
    - Each function or method is a separate controller that can now call different models and views.
  + Lines 19 to 35: are definitions of various functions (methods), i.e. controllers in this case, which we need to write.
    - The home controller is same as before. We can see in the home() function that we ae getting data from the model3D\_info() model: $data = $this->model->model3D\_info();
    - And, we are loading the HTML view called ‘view3DAppTest\_2’ and passing the retrieved $data to it. This is a Model, View Controller design pattern, although very much simplified, in action. You can see that we could use this approach to get data from any other model or pass it to any other view — separation of model (the data) from the view (the web page) via the controller.
    - Each other controller at the moment just echo’s a string to the browser. We will define the body of these controllers soon, but before we move further let’s check our PHP code works ok.
    - It stands to reason, if you enter a name for a controller after index.php, then that controller must exist. Note, at the moment we have no error trapping, so if you type something like …/index.php/foobar, then a PHP fatal error will occur, see Figure 20: The foobar controller method doesn’t exist!.

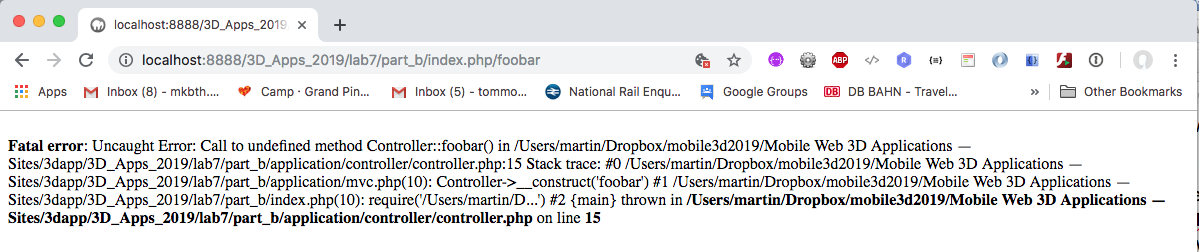
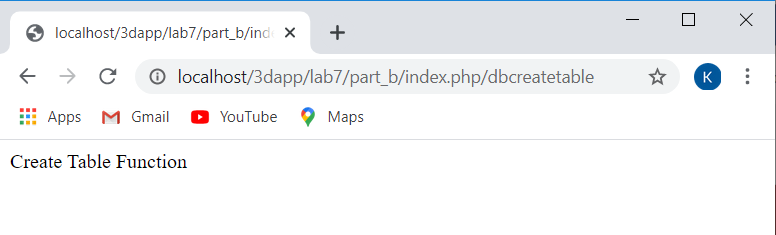


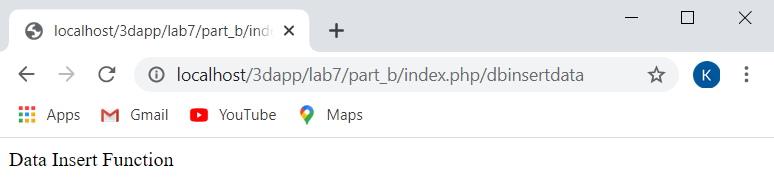
Figure 20: The foobar controller method doesn’t exist!

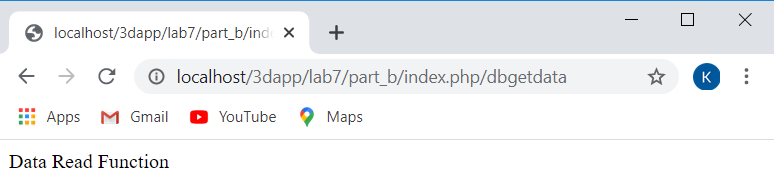
* + - So, in this example the value of the ‘pageURI’ variable is the name of a Controller class method. The new Controller($pageURI); php code) will call the controller constructor methods dbcreatetable, dbinsertdata and dbgetdata.
  + If the user types no controller name, e.g. if a user types: <http://localhost:8888/3D_Apps_2019/lab7/part_b/index.php/>   
    the home controller will be called as a default controller — new Controller('home');

## Test your controller code

1. Run your code by typing URL with four different parameters, as shown in Figure 21 below:







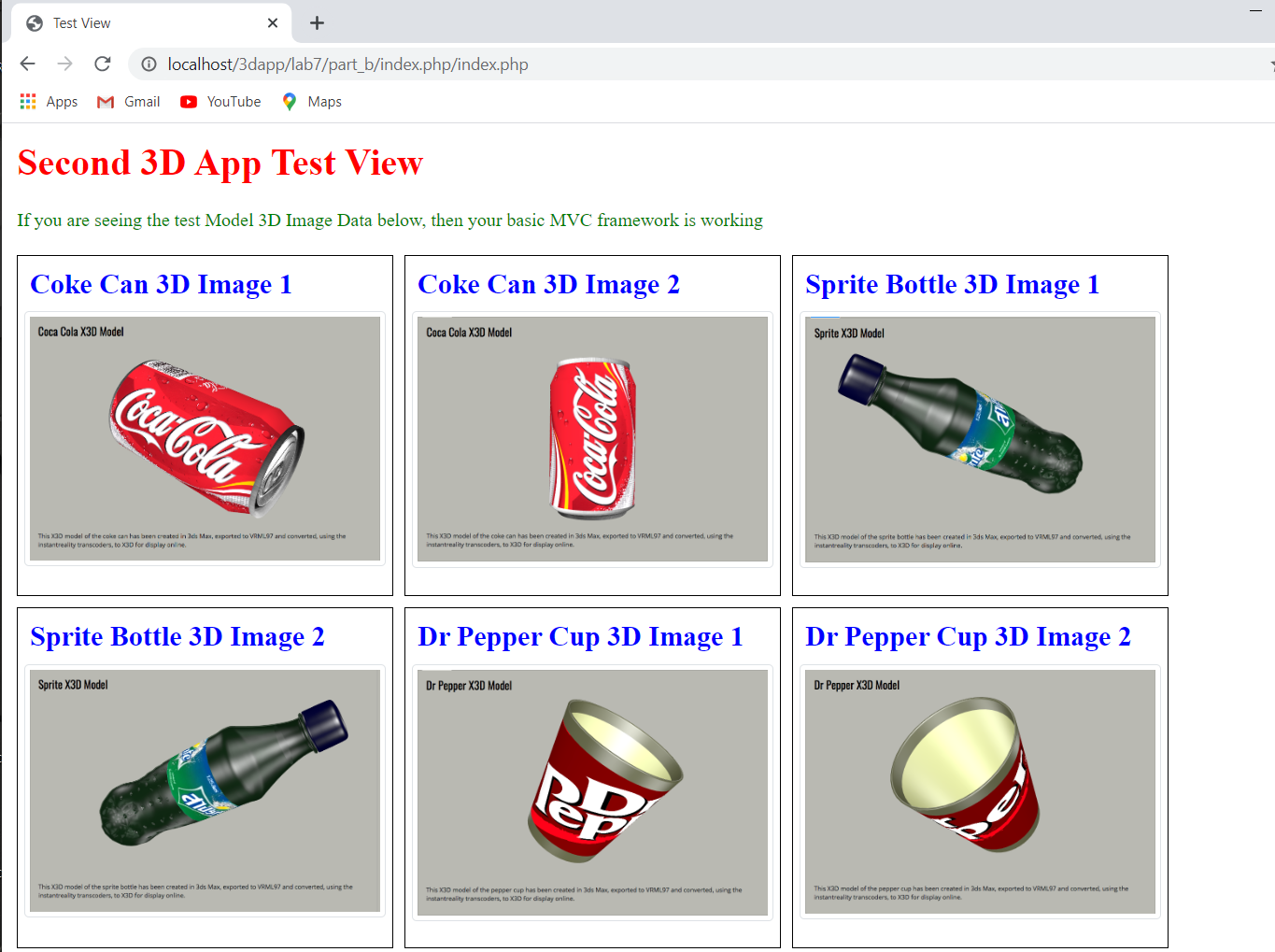


Figure 21: Accessing the controller methods via the URI

Remember: the url above is ‘http://localhost/lab7/part\_b/index.php/’, since I am running in a localhost, but it will be different in your case. If you are using public\_html folder on your ITS hosting space, your URL should be something like this:

user.sussex.ac.uk/~your-user-name/3dapp/lab7/part\_b/index.php/*class\_method*

At this point, if you see the results as above, then you are doing fine and can carry on further below. If you are getting errors, then fix them before you proceed further.

## Connect up the SQLite database

1. Edit your model.php file in your ‘/application/model/’ folder to include a $dbhandle variable and assign a database object to it, see Figure 22 for the PHP code to do this. You are creating a new PDO (PHP data Object) object that represents a connection to a database.

* This model.php file offers a level of abstraction on top of the chosen database, which is SQLite, by providing a simple PHP PDO wrapper class. This wrapper class can be used to get started with PDO by providing common methods you might to need to get going.
* By using this PDO wrapper class, we are effectively using two layers of abstraction, the class itself and PDO.
* If you search the Internet carefully, you will find many useful resources where they provide similar, maybe even better PDO wrapper classes, here is an interesting tutorial that delivers the same thing: <http://culttt.com/2012/09/24/prevent-php-sql-injection-with-pdo-prepared-statements/> and provides another discussion on Preventing PHP SQL Injection with PDO Prepared Statements.
* Here is another interesting tutorial: <http://code.tutsplus.com/tutorials/why-you-should-be-using-phps-pdo-for-database-access--net-12059>

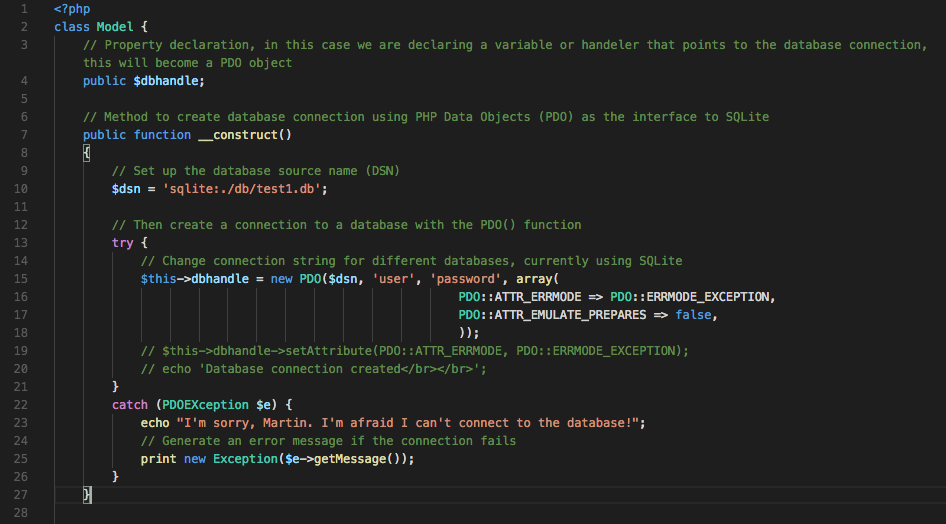


Figure 22: Update the model to provide database connectivity

* You should always use exceptions to handle error by wrapping your PDO in a try/catch block where the catch is used to process the error message. PDO can be forced into an error mode, in this case we are specifying the PDO::ERRMODE\_EXCEPTION, which is used in most situations. You can use the setattribute() to set the error mode, or input it as an element in the array as the fourth attribute in the PDO() function above.
* And remember, use the PDO::ATTR\_EMULATE\_PREPARES => false to guard against [SQL injection](https://www.esecurityplanet.com/threats/what-is-sql-injection.html).

1. Code explanation:
   * Line 2 creates the class Model, which as we discussed above is an object, and an object has methods (functions) and properties (constants, variables) that we can manipulate.
   * Line 4: contains declaration of a public variable named ‘dbhandle’
   * Lines 6 to 27: we now have a constructor for the model class that creates a PDO object, remember a constructor is the default method, which in this case creates a connection to a database.
   * Line 10: we create a variable for the database source name (dsn).
   * Line 15 to 18: in the try block assigns the new PDO (PHP Database Object) to the $dbhandle variable.
     + The PDO object first specifies a database connection string (dsn) starting with the type of database, in this case SQLite, followed by a path to the database and database name, which in this case the path is the db folder and the database name is test1.db – this is referred to as a DSN or database source name.
     + If you now wanted to change the database to another database, e.g. mySQl, you would only need to change the DSN.
     + Next you pass in the user name (user) and password (password) in this case.
     + For development learning, it is a good idea to set the PDO error mode attribute PDO::ERRMODE\_EXCEPTION, which throws an exception if there is an error. Also, to prevent SQL injection attacks you must set the PDO::EMULATE\_PREPARES to false. Both attributes can be set by passing them into the new PDO function as an array for the fourth parameter.
   * Line 19: commented out is another way to set the PDO attributes mentioned above.
   * Line 20: commented out just echo’s that you have connected to the database for debugging
   * Line 23 to 25: If the connection to the database fails PDO will throw a PDOException, which can be trapped in the catch part of the try/catch block, and we echo out a meaningful message

If all goes well, this should provide a successful connection to the SQLite database!

1. So, we now need to create the three database connectivity functions we discussed above to add database functionality in our MVC framework. The three new functions are: dbCreateTable, dbInsertData and dbGetData. Add these functions inside your model class just before last closing parentheses ‘}’ (i.e. after the body of car\_info method).
   * Start by adding the dbCreateTable function, as shown in Figure 23.

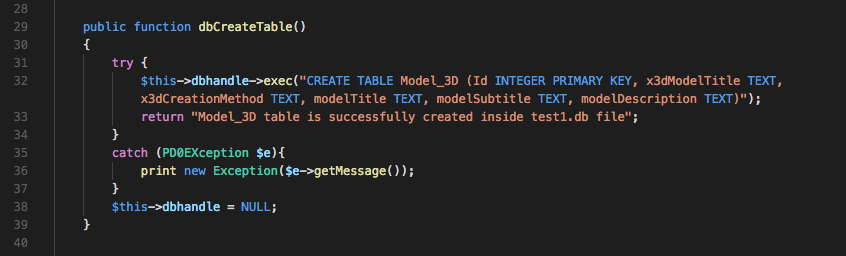


Figure 23: Creating a table in the database using the PDO exec function.

* Code explanation:
  + Line 32: The PDO::exec() function executes an SQL statement in a single function call, where in this case the SQL statement is designed to create a table in the database. The SQL query string creates a table named ‘Model\_3D’ with several field names:
    - ‘Id’ field: an integer primary key field, which is automatically incremented, when we enter any data in all other fields. So, its default values would be 1, 2, 3, 4, and so on.
    - ‘x3dModelTitle’ field: can store any string, but it cannot be left empty. We will use this field to store the title of your X3D model.
    - ‘x3dCreation Method’ field: can also store string values, we will use this field to store a short description of how you created the X3D model.
    - ‘modelTitle’ field: another text (i.e. string) field for storing the Title in the description .card class of X3D model
    - modelSubtitle’ field: another text (i.e. string) field for storing the Subtitle in the description .card class of X3D model
    - modelDescription’ field: another text (i.e. string) field for storing the Description in the description .card class of X3D model
    - You will have observed that I have chosen these as examples taken from the data.json file in Lab 6. This is a very simple database back-end for the model contents, e.g. coke, sprite and pepper models. So, I have only given here an example table to cover the X3D model card and the description card, which I have ‘lumped’ into the one table. Later, you might want to create separate tables for each of the cards, or a single table for all the dynamic data on the coke, sprite or pepper pages. This is a database design decision that you can make later.
    - Incidentally, if you include titles, subtitles, and associated texts for all the cards, I count 13 pieces of dynamic data of type TEXT for each of the coke, sprite and pepper contents — at this stage of the tutorial I am only creating a simple database table for 4 pieces of dynamic data.
  + Line 35 to 37: If the table has not been not created the PDO exec function would throw an error, which is trapped in the catch block.
  + Line 38: closes the database connection, which was opened in the \_\_constructor of the model class.
  + Lastly line 33: returns a success message to the function calling program, i.e. the controller.

1. Now add the second function named ‘dbInsertData’, immediately after ‘dbCreateTable’ function. The code for dbInsertData is given below, which is similar in format to the dbCreateTable function:



Figure 24: The dbInsertData function

* Code explanation
  + Line 42: The PDO::exec() function executes an SQL statement in a single function call, where in this case the SQL statement is designed to insert data into the database table. The SQL query string inserts values for 5 fields discussed above:
    - Note: We did not have to define the Id field in the original table definition inside ‘dbCreateTable’ function; we could have set it to generate auto numeric values.
    - We can see in lines 45 to 50 we are inserting the values, and temporarily I am just sticking in test string values. If, and when it works, you should insert appropriate string values taken from your data.json value
  + Line 56: We close the database connection, which was opened in \_\_constructor of the model class.
  + Line 51: Returns a success message to the function calling program.

1. Next, we need to create the function named ‘dbGetData’ to retrieve data from database. You should add the following function code, see Figure 25, inside your model class, after the body of ‘dbInsertData’ function defined above.

* This function is quite interesting because from this we can develop several variations that you could cast into separate functions for getting: single values, a particular value, all results with a for loop, printing all results to a table, using a while loop to get all values and returning the results to a calling function.

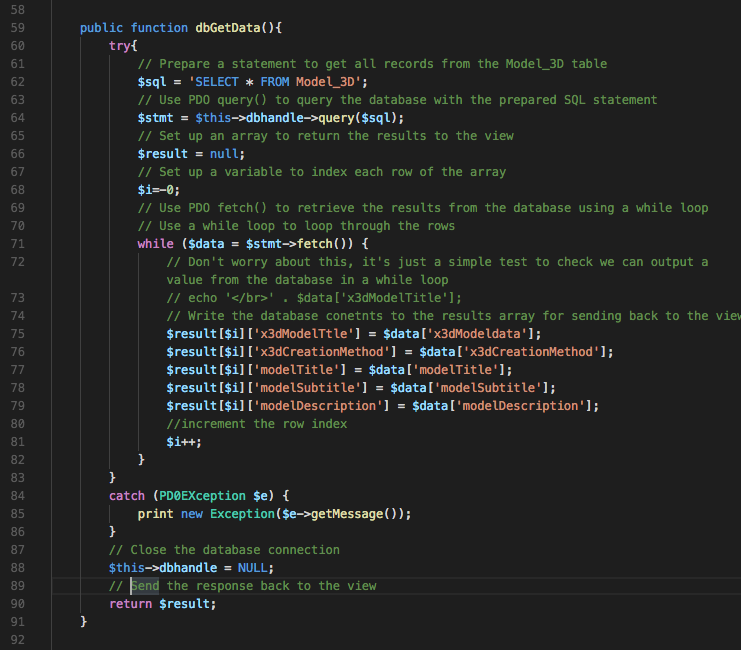


Figure 25: The dbGetData function

* Code explanation
  + Lines 62: sets up an SQL query to retrieve all data from the Model\_3D table
  + Line 64: PDO::query() executes this SQL statement in a single function call, returning the result set (if any) as a PDO statement object, $stmt.
  + Line 66: declares a result array.
  + Line 68: Sets up an incrementor variable for the while loop.
  + Line 71 to 82: contains a while loop, that repeats its body for the number rows found in database through the query.
    - The PDO::Statement::Fetch fetchs a row from a result set associated with a PDOStatement object.
    - The values for each row are stored in an array named ‘result’, so that we can easily pass it to the ‘view’ for displaying on the screen.
  + Line 88: closes the database connection, which was opened in \_\_constructor of the model class.
  + Line 98: returns an array containing values of all data from the database.

1. Now modify the controller class methods to add definitions for our previously tested functions, see Figure 26: dbCreateTable, and dbInsertData and dbGetData.



Figure 26: Updated controller functions

* Looking at Figure 26 you will see that, rather than replace the db functions, I have added new methods starting with ‘api’. This is simply to better illustrate the separation between the controller calling methods and the database methods. If we use the term api as part of the controller method name, e.g. apiCreateTable, it emphasises these are different methods. The PHP ‘api’ methods belong to the Controller class and the PHP ‘db’ methods belong to the Model class.
* Code explanation
  + Lines 24 to 29: is a new function ‘apiCreateTable’ in the controller — as mentioned, this is a method in the Controller class. This is the method we will call in the URL, and later in Part C we will use the JQuery .getJSON() AJAX method to make these api calls. This method invokes another method in the Model class, which is called dbCreateTable.
    - So, apiCreateTable() used the current instance of the dbCreateTable() method (or function) from the Model class, and the returning values are stored in ‘data’ variable.
    - Recall: we have defined this dbCreateTable() function (or method) in the Model class; a call to this function will create a table in the SQLite database.
    - Note: if you use call this function once, a file named ‘test1.db’ will be created inside the ‘db’ sub-folder of the part\_b folder. If you want to create the table again then either delete the ‘test1.db’ file or edit your model class change its name to something new, otherwise you will get an error that database already exists!
    - Next, we call the view method from the ‘Load’ class with parameters ‘viewMessage and ‘data’ array.
    - Recall: we defined the ‘Load’ class previously in ‘/application/view/load.php’ file. A call to Load view method (or function) in this case will load the ‘viewMessage.php’ file as the current view.
    - The viewMessage.php file simply uses the PHP echo() function to output (or echo) the data that is returned from the dbCreateTable or dbInsertData methods (or functions) in the Model class, see Figure 27. In this case, the returned data from dbCreateTable or dbInsertdata is a simple message form their associated ‘try’ blocks.
  + Lines 38 to 34: contains the definition of ‘apiInsertData’ function. This function:
    - Inputs the data into the table specified inside the function (table name is ‘Model\_3D’ in this case)
    - Next, we call the view function from the ‘Load’ class with parameters ‘viewMessage and ‘data’ array, as we did for the dbCreateTable function.
    - Note: if you run this function many times, your table will be populated with duplicate data records. If this happens and you want to clean your table, then delete ‘test1.db’ database file, create table and insert data again.
  + Lines 35 to 39: contains the definition of the ‘apiGetData’ method. This function:
    - Calls the ‘dbGetData’ function from the model class, retrieve data from the‘Model\_3D’ table
    - On the next line, we again call the view function from ‘Load’ class, but this time with the parameters ‘view3DAppData’ and ‘data’ array to generate a different front-end view. The view function this time will call ‘view3DAppData.php’ file to generate a view for data array.

That completes the controller for now. It is interesting to think of the calls to the controller methods as simple API calls, which in effect invoke the PHP Model class methods. The Model class can be further thought of as a Data Access Layer — in fact, that is exactly what PDO ([PHP Data Objects](http://php.net/manual/en/intro.pdo.php)) is, a lightweight, consistent interface for accessing databases in PHP.

## Develop our views

1. Create a new PHP file named ‘viewMessage.php’ inside ‘/application/view’ folder and add the following code to it, see Figure 27.

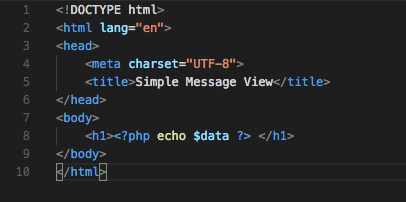


Figure 27: The simple message function

* Code explanation
  + Line 8: we have displayed value of data variable inside ‘<h1>’ tag.
    - This file is being called by the controller in two functions:
    - apiCreateTable, which uses the load function’s view method to display in the current view the data returned from the dbCreateTable method in the Model class. The data in this case is:
      * "Model\_3D table is successfully created inside test1.db file"
    - apiInsertData, which uses the load function’s view method to display in the current view the data returned from the dbInsertData method in the Model class. The data this case is:
      * "X3D model data inserted successfully inside test1.db"

1. Create another file named ‘view3DAppdata.php’ inside ‘/application/view’ folder and add following code in it, see Figure 28. Clearly, we are creating another view, which could be one of many views in the Model, View, Controller design pattern.



Figure 28: The car view HTML code with data inserted from PHP

* Code explanation
  + This view file is being called by the Controller class apiGetData function (or method)
  + I think that you can easily work out what this view does. We simply loop round the $data array returned from the Model class via the controller and echo out this data into some appropriate HTML formatted by CSS.

## Test your new MVC code

1. When you first open the index.php in a browser, the database base connection is automatically made because of the way we have organised the MVC framework. So, the first thing we need to do is create a table and define its fields by calling the ‘apiCreateTable’ method in the Controller class from our framework, see Figure 29.

* You should see the output as below, if you encounter any error then the most likely places to fix these are in the controller.php file or the model.php file

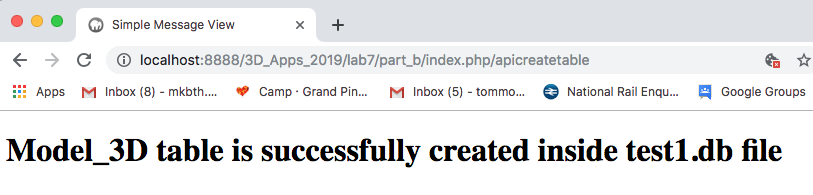


Figure 29: A new database has been created with a Model\_3D table

Note: the url above is ‘http://localhost/lab7/part\_b /index.php/dbcreatetable, since I am running my code in localhost, but it will be different in your case. If you are using public\_html folder on your ITS hosting space, your URL should be something like this:

users.sussex.ac.uk/~your-user-name/3dapp/lab7/part\_b/index.php/apicreatetable

Note 2: If you want to run the ‘apiCreateTable’ controller method again, then before you do run it again make sure you delete ‘test1.db’ file from the ‘/db/’ folder. If you don’t delete the test1.db file and invoke this method more than once, it will try to create another Model\_3D table in the ‘test1.db’ and you will encounter an error message.

Also, it is worth looking back to Lab 6 where we used the JQuery .getJSON() function to make an AJAX call to the backend. You should be able to realise that we could simply use this URL in the .getJSON(); we need to return JSON data of course.

1. Next insert some data into your table by calling ‘apiInsertData’ controller, see Figure 30.

* You should see the output as below, if you encounter any error then the most likely places to fix these are:
  + controller.php file
  + model.php file

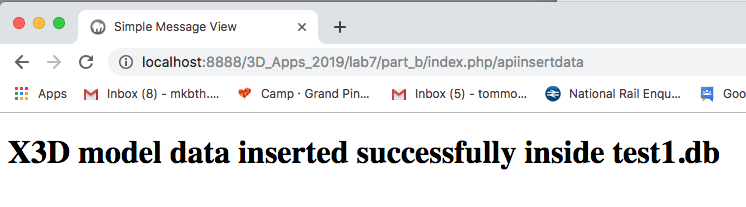


Figure 30: Data has been inserted into the table

1. Next, we call ‘apiGetData’ method in the controller to see the data from SQLite database, see Figure 31.

* You should see the output as below, if you encounter any error then the most likely places to fix these are:
  + controller.php file
  + model.php file

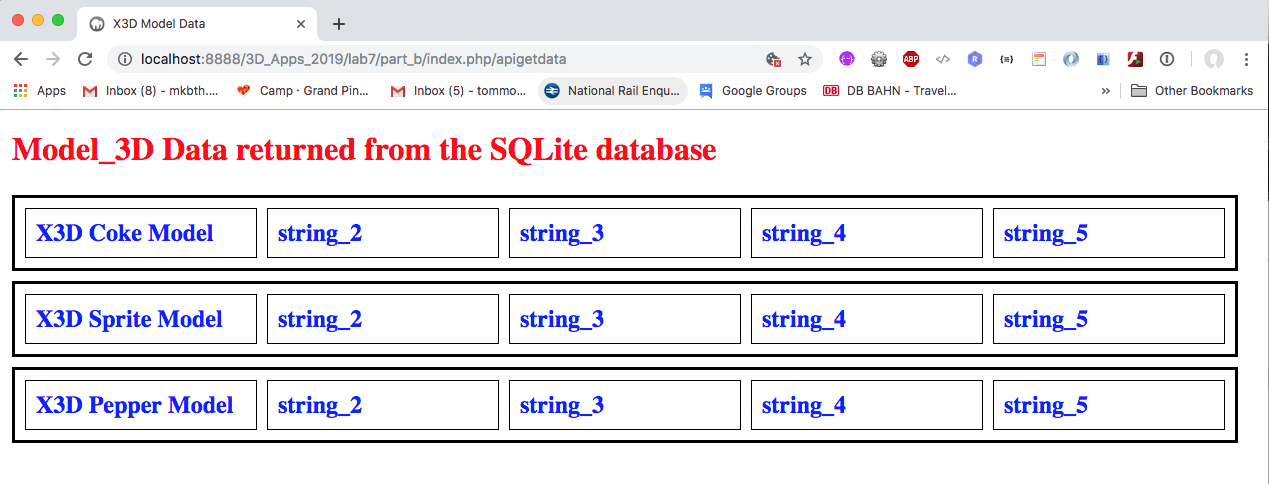


Figure 31: apiGetdata is used to get the database contents and display in an HTML table

* Clearly, you can see here that I have started to put the actual data from the Lab 6 data.json file into the dbInsertTable method to illustrate.

## Some improvements

It would be a good idea to think about creating more controller functions to manipulate the database based on typical CRUD (Create, Read, Update and Delete) functionalities. For example, think about building controller functions for:

* Inserting a new field: (i.e. the remaining data elements in your Lab 6 view) as well as other properties
* Updating a field
* Getting a single value from a field
* Getting a single field
* Getting all fields of the same property, e.g. modelDescriptions.
* …

Finally, currently the MVC architecture is quite simple. It returns an array from the database that is parsed into a table in the view. How can you write a model function that returns a JSON file, which is then parsed by the controller for the view? MVC Part C will develop this further.

# Laboratory 8

In Lab 8 you will continue with Part C of this tutorial — MVC Design Pattern with JavaScript, JQuery, AJAX, JSON, PHP and SQLite.

The main thing to take out of Part C will be connecting the front end views via AJAX using the JQuery .getJSON() function, like you did in Lab 6, using the Controller class methods to connect to the Model class method and so retrieve data asynchronously from the backend server.

1. http://php.net/manual/en/book.pdo.php [↑](#footnote-ref-1)